Systematic Extraction and Implementation of Divide-and-Conquer Parallelism

Sergei Gorlatch

University of Passau, D–94030 Passau, Germany

Abstract. Homomorphisms are functions that match the divide-and-conquer paradigm and thus can be computed in parallel. Two problems are studied for homomorphisms on lists: (1) parallelism *extraction*: finding a homomorphic representation of a given function; (2) parallelism *implementation*: deriving an efficient parallel program that computes the function. A systematic approach to parallelism extraction proceeds by generalization of two sequential representations based on traditional *cons* lists and dual *snoc* lists. For some non-homomorphic functions, e.g., the maximum segment sum problem, our method provides an embedding into a homomorphism. The implementation is addressed by introducing a subclass of distributable homomorphisms and deriving for them a parallel program schema, which is time optimal on the hypercube architecture. The derivation is based on equational reasoning in the Bird-Meertens formalism, which guarantees the correctness of the parallel target program. The approach is illustrated with function *scan* (parallel prefix), for which the combination of our two systematic methods yields the "folklore" hypercube algorithm, usually presented *ad hoc* in the literature.

1 Motivation and Notation

The problem of programming parallel machines can be managed if put on a solid formal basis, which allows to address both correctness and performance issues during the design process, rather than as an afterthought. This paper deals with divide-and-conquer parallelism by studying functions called homomorphisms.

Definition 1. A function h defined on lists is a *homomorphism* iff there exists a binary associative operation $\circledast$ such that, for all lists x and y:

$$h(x +\!\!+\, y) = h(x) \circledast h(y) \qquad (1)$$

where $+\!\!+$ is the list concatenation.

Intuitively this means that the value of h on the concatenated list depends in a particular way, using the *combine operation* $\circledast$, on the values of h applied to the pieces of the list. The computations of $h(x)$ and $h(y)$ are independent and can be carried out in parallel, so (1) can be viewed as expressing the well-known divide-and-conquer paradigm. Examples of homomorphisms are simple functions, such as summing up the elements of a list of numbers, and also more complicated and important functions like *scan* (prefix sums) [5].

In the Bird-Meertens formalism (BMF) [4, 26], functions including homomorphisms are defined on arbitrary composite types (trees, arrays, etc.), and equational reasoning is used for deriving programs by transformation [29]. The use of higher-order functions has been popular in the data-parallel setting [18].

We restrict ourselves to non-empty lists and use the following notation (for brevity, definitions are informal):

$[\alpha]$	the type of lists whose elements are of type α;
$+\!\!\!+$	list concatenation;
$\circ$	backward functional composition;
$map\ f$	*map* of an unary function f, i.e. $map\ f\ [x_1, \ldots, x_n] = [fx_1, \ldots, fx_n]$;
$red\,(\odot)$	*reduce* with a binary associative operation $\odot : \alpha \to \alpha \to \alpha$, $red\,(\odot)\,[x_1, \ldots, x_n] = x_1 \odot x_2 \odot \ldots \odot x_n$;
$< >$	Backus' FP *construction*: $< f_1, \ldots, f_n > x = (f_1\,x, \ldots, f_n\,x)$;
$zip\,(\odot)$	combines elements of two lists of equal length with operation $\odot$, $zip\,(\odot)\ ([x_1, \ldots, x_n], [y_1, \ldots, y_n]) = [(x_1 \odot y_1), \ldots, (x_n \odot y_n)]$.

Theorem 2 (Bird [4]). *Function h is a homomorphism iff it can be factored into the composition:*

$$h = red\,(\circledast) \circ (map\,f) \tag{2}$$

We write $hom\,(f, \circledast)$ for the unique homomorphism with combine operation $\circledast$, such that $h\,([a]) = f\,(a)$, for all a. The theorem provides a standard parallelization pattern for all homomorphisms as a composition of two *stages* [13]. Whereas the first stage in (2), *map*, is totally parallel, the reduction can be computed in parallel on a tree-like structure, with $\circledast$ applied in the nodes.

There are two main problems with homomorphisms:

• *Parallelism extraction.* It would be desirable to be able to extract the homomorphic parallelism of a given function h, i.e., to find the corresponding combine operation $\circledast$ satisfying property (1). For functions like *length* this construction is simple, but already for the *scan* function it requires a formal correctness proof [23] or intuition [15]. For non-homomorphic functions, the problem is how to "massage" them into a homomorphism [7].

• *Parallelism implementation.* The reduction stage of (2) may be inefficient: e.g., for functions which yield lists, its direct implementation has linear time complexity because of communication, and this cannot be improved by increasing the number of processor [27].

We propose systematic approaches to both problems and present them as follows. In Section 2, we consider two sequential functional representations, based on *cons* and *snoc* lists, and show that their generalization as terms can be used for extracting a homomorphic representation. Section 3 extends this method to non-homomorphic functions and illustrates it for a well-known problem, maximum segment sum. In Section 4, we consider homomorphisms whose direct parallel implementation suffers from high communication costs and introduce a subclass, called DH (distributable homomorphisms), for which these costs can be cut down. A common parallel implementation schema for DH is derived and mapped onto a hypercube in Section 5. We illustrate with the *scan* function: its homomorphic representation is extracted in Section 2; then, in Section 6, it is systematically adjusted to the DH-format and implemented on the hypercube, yielding the "folklore" algorithm, which is usually presented in an *ad hoc* manner [25]. We compare to the related work in Section 7 and then conclude.

2 Extracting Homomorphisms

We restrict ourselves to finite non-empty lists. Whereas homomorphic representations use list concatenation, traditional functional programming is based on the constructors *cons* and *snoc*. We use $\cdot\!:$ for *cons*, which attaches an element at the front of the list, and $:\!\cdot$ for *snoc*, which attaches the element at the list's end.

Definition 3. List function h is called *leftwards* (*lw*) iff there exists a binary operation $\oplus$, such that $h\,(a \cdot\!: y) = a \oplus h(y)$ for all elements a and lists y. Dually, function h is *rightwards* (*rw*) iff, for some $\otimes$, $h\,(x :\!\cdot\, b) = h(x) \otimes b$.

Since $\oplus$ and $\otimes$ may be complicated, many functions are either *lw* or *rw* or both. Following fact was proved by Meertens and presented systematically by Gibbons [9, 11].

Theorem 4. *Function h is a homomorphism iff it is leftwards and rightwards.*

Unfortunately, as pointed out in [11], the theorem does not provide a way to construct the homomorphic representation of a function from its *lw* and *rw* terms. We try to rectify this by introducing a new definition.

Definition 5. Function h is called *left-homomorphic* (*lh*) iff there exists (possibly non-associative) combine operation $\oplus$, such that $h\,(a \cdot\!: y) = h\,([a]) \oplus h\,(y)$. The dual definition of *right-homomorphic* (*rh*) function is obvious.

Every *lh* (*rh*) function is also *lw* (*rw*), but, e.g., function g is *lw* but not *lh*:

$$
\begin{aligned}
g\,[a] &= |a| \\
g\,(a \cdot\!: y) &= \texttt{if}\quad a \le g\,(y)\quad \texttt{then}\quad |a + g\,(y)|\quad \texttt{else}\quad |a - g\,(y)|
\end{aligned}
$$

Theorem 6. *If function h is a homomorphism with combine operation $\circledast$, i.e., $h = \mathrm{hom}\,(f, \circledast)$, then h is both lh and rh with the same combine operation. If function h is lh or rh, and the combine operation is associative, then h is a homomorphism with this combine operation.*

Proof: see [12], where we prove a slightly stronger proposition.

The following example (courtesy of J. Gibbons) shows why the test for associativity in the second part of the theorem is necessary. The identity function *id* on lists can be defined as both *lh* and *rh* with combine operation: $u \circledast v = [\mathit{head}\,u] \mathbin{+\!\!+} \mathit{init}\,v \mathbin{+\!\!+} \mathit{tail}\,u \mathbin{+\!\!+} [\mathit{last}\,v]$, but it is clearly not a homomorphism with this operation.

Theorem 6 suggests a possible way to find a homomorphic representation: construct a *cons* definition of the function in the *lh* format (or, dually, find an *rh* representation on *snoc* lists) and prove that the combine operation is associative. Sometimes this simple method works, as the following example of function *length*, computing the length of a list, demonstrates.

Example 1. Since $length\,([a]) = 1$, we have $f = one$, where $one\,(x) = 1$. The *cons*-definition: $length\,(a \cdot\!: y) = 1 + length\,(y) = length\,([a]) + length\,(y)$. Thus *length* is *lh*. From Theorem 6 and associativity of $+$: $length = hom\,(one, +)$.

Our next example demonstrates that the method with *lh*/*rh* representations does not always go that smoothly.

Example 2. A more complicated example is function *scan* which, for an associative operation $\odot$ and a list, returns the list of the "running totals" with $\odot$, e.g.: $scan\,(\odot)\,([a, b, c, d]) = [a,\ a \odot b,\ a \odot b \odot c,\ a \odot b \odot c \odot d]$.

The sequential *cons* definition of *scan* is as follows:

$$scan\,(\odot)\,(a \cdot\!: y) \;=\; a \;\cdot\!:\; (map\,(a\,\odot)\,(scan\,(\odot)\,y)) \tag{3}$$

Here, so-called sectioning is exploited in that we fix one argument of $\odot$ and obtain the unary function $(a\,\odot)$, which can be *map*ped.

Representation (3) does not match the *lh* format because a is used where only $scan[a]$ is allowed. Since $scan\,[a] = [a]$, there are different possibilities to express a via $scan[a]$, e.g., $a = fst\,(scan[a])$ or $a = last\,(scan[a])$, or we could use $((scan[a])+\!\!+)$ for $(a\cdot\!:)$. Thus we obtain six possible terms for $\circledast$; however, none of these terms defines an associative operation!

Let us try to use rightwards-homomorphy in the *snoc* definition:

$$scan\,(\odot)\,(x :\!\cdot\, b) \;=\; (scan\,(\odot)\,x) \;:\!\cdot\; (last\,(scan\,(\odot)\,x) \odot b) \tag{4}$$

Alas, we run into a similar problem as for *cons* lists: both obvious substitutions for b, namely $fst\,(scan\,[b])$ and $last\,(scan\,[b])$, lead to a non-associative operation, and thus we are still not able to express the *scan*-function as a homomorphism.

We proceed with general considerations and then get back to the example. According to Definition 1, there is a term TH over $h\,(x)$ and $h\,(y)$ that defines an associative operation: $TH\ :\ h\,(x) \circledast h\,(y)$. For term T, let $T.\{u \mapsto c\}$ denotes the result of substituting c for variable u in T. The following two terms, built from TH by substitutions: $TL = TH.\{x \mapsto [a]\}$ and $TR = TH.\{y \mapsto [b]\}$, are obviously in the *lh* and *rh* format (see Definition 5), correspondingly.

Terms TL and TR are semantically equivalent to all *cons* and *snoc* representations of function h, correspondingly. In Example 2, we have unsuccessfully tried to pick a *cons* term TC or a *snoc* term TS and to transform them into the desired format using various equalities of the theory of lists. In the following, $T_1 \overset{E}{\rightleftharpoons} T_2$ means that terms T_1 and T_2 are semantically equivalent in equational theory E.

Relations between the terms introduced so far are illustrated in the following diagram by solid lines:

$$\begin{tikzcd}
& TL : h\,[a] \circledast h\,(y) \arrow[rr, shift left, "E"] & & TC \arrow[ll, shift left] \arrow[d, dotted, "lw"] \\
TH : h\,(x) \circledast h\,(y) \arrow[ur, "\{x \mapsto [a]\}"] \arrow[dr, "\{y \mapsto [b]\}"'] & & TG \arrow[ll, dotted, "Assoc"'] & \boxed{\text{Generalization}} \arrow[l, dotted] \\
& TR : h\,(x) \circledast h\,[b] \arrow[rr, shift left, "E"] & & TS \arrow[ll, shift left] \arrow[u, dotted, "rw"']
\end{tikzcd}$$

Our ultimate goal is to find term TH from given TC and TS, e.g., for *scan* function, TC and TS are the right-hand sides of (3) and (4), correspondingly. We use so-called *generalization* or *anti-unification* of terms [16].

Definition 7. *Generalization* in equational theory E of terms T_1 and T_2 w.r.t. substitutions σ_1 and σ_2 is term $TG = Gen_E\{T_1 \,|\, \sigma_1, T_2 \,|\, \sigma_2\}$, which satisfies $TG.\{\sigma_1\} \overset{E}{\rightleftharpoons} T_1$ and $TG.\{\sigma_2\} \overset{E}{\rightleftharpoons} T_2$.

Theorem 8. *If an lw term TC and an rw term TS, both for function h, are generalized to the term $Gen_E\{TC \,|\, \{x \mapsto [a]\}, TS \,|\, \{y \mapsto [b]\}\}$, which defines an associative operation $\circledast$, then h is a homomorphism with $\circledast$ as combine operation.*

We call the generalization in Theorem 8 the *CS-Generalization* (CS for "Cons + Snoc") in the theory of lists E. The envisaged CS method of finding the combine operation can be understood from the above diagram by moving along the dotted arrows. Two terms provided by the user, TC for a *cons* and TS for a *snoc* representation of function h, are first checked to be in the lw/rw format, correspondingly. If so, they are CS-generalized to term TG, and then associativity of the operation defined by TG is checked.

For function *scan*, representation (3) is obviously *lw* and (4) is *rw*. Their CS-generalization yields term

$$TG \;=\; scan\,(\odot)\, x \;+\!\!\!+\; (\, map \;\; (last\,(scan\,(\odot)\, x) \odot) \;\; (scan\,(\odot)\, y)\,)$$

The operation $\circledast$ defined by TG is associative:

$$u \circledast v \;=\; u \;+\!\!\!+\; map\,(last\,(u) \odot)\, v \tag{5}$$

Therefore, *scan* is a homomorphism: $scan\,(\odot) \;=\; hom\,([.], \circledast)$, with $\circledast$ from (5).

Designing a CS-generalization procedure and investigating its properties is our present topic of research, which is beyond the scope of this paper.

The next section shows that the CS approach is even more useful in case of non-homomorphic functions.

3 Almost-Homomorphisms

Many practical non-homomorphic functions are so-called *almost-homomorphism*: they are convertible to a composition of a homomorphism and some adjusting function. Cole [7] reports several case studies on constructing a homomorphism as a tuple of functions, where the original function is one of the components. [1]

The main difficulty is to guess, which *auxiliary functions* must be included in a tuple and then to find the corresponding combine operation. Usually, this requires a lot of ingenuity from the program developer. We show that the "cons + snoc" approach allows to construct tuple-homomorphisms systematically.

Example 3. We consider the *maximum segment sum* (*mss*) problem – a *programming pearl* [3], which has been studied by many authors [4, 7, 26, 28].

Given a list of integers, function *mss* finds the contiguous segment of the list whose members have the largest sum among all such segments and returns this sum, e.g., in the notation of [7]:

$$mss\,[\,2,-4,2,-1,6,-3\,]=7$$

where the result is contributed by the segment $[2,-1,6]$. The empty segment is defined to have sum 0, so the result is always non-negative.

Let us first express function *mss* over *cons* lists. For some element a and list y, it may well be the case that $mss\,(a \cdot\!: y) = (mss\,[a]) \uparrow (mss\,y)$, where $\uparrow$ returns the larger of its two integer arguments. But we must not overlook the possibility that the true segment of interest includes both a and some (initial) segment of y. Therefore, we have to introduce an auxiliary function *mis* which yields the sum of the *maximum initial segment*. We add the definition of *mis* and obtain the following closed definition of the tuple function $< mss, mis >$ on *cons* lists:

$$\begin{aligned} mss\,(a \cdot\!: y) &= mss\,[a] \uparrow mss\,y \uparrow (a + mis\,y)\\ mis\,(a \cdot\!: y) &= mis\,[a] \uparrow (a + mis\,y) \end{aligned}$$

Our approach requires to define *mss* on *snoc* lists. This leads to another auxiliary function, *mcs*, yielding the sum of the *maximum concluding segment*:

$$\begin{aligned} mss\,(x :\!\cdot\, b) &= mss\,x \uparrow (mcs\,x + b) \uparrow mss\,[b]\\ mcs\,(x :\!\cdot\, b) &= (mcs\,x + b) \uparrow mcs\,[b] \end{aligned}$$

The introduction of tuples requires the following natural extension of the CS method: (1) the notion of homomorphism is straightforwardly extended for tuples; (2) generalization works for representations of the *union tuple* which in the *mss*-example is: $< mss, mis > \cup < mss, mcs > = < mss, mis, mcs >$.

Trying to find a *cons*-definition for *mcs* and a *snoc*-definition for *mis*, we see that, e.g., the concluding segment of $(a \cdot\!: y)$ may be the whole list, so we need to know its sum, which no (combination) of the three functions can yield. We have to introduce one more auxiliary function *ts* (for *total sum*).

[1] Actually, every function can be made "homomorphic" by tupling with the identity function, however, this trivial case is clearly of no interest for parallelization.

The constructed quadruple $<mss, mis, mcs, ts>$ has following *cons* and *snoc* representations which are obviously *lw* and *rw*, correspondingly:

$$
\begin{array}{rl|rl}
mss\,(a \cdot\!: y) = & mss\,[a] \uparrow (a + mis\,y) \uparrow mss\,y & mss\,(x :\!\cdot\, b) = & mss\,x \uparrow (mcs\,x + b) \uparrow mss\,[b] \\
mis\,(a \cdot\!: y) = & mis\,[a] \uparrow (a + mis\,y) & mis\,(x :\!\cdot\, b) = & mis\,x \uparrow (ts\,x + b) \\
mcs\,(a \cdot\!: y) = & mcs\,y \uparrow (a + ts\,y) & mcs\,(x :\!\cdot\, b) = & mcs\,[b] \uparrow (mcs\,x + ts\,[b]) \\
ts\,(a \cdot\!: y) = & ts\,[a] + ts\,y & ts\,(x :\!\cdot\, b) = & ts\,x + ts\,[b]
\end{array}
$$

After applying the CS-generalization procedure pair-wise (see [12] for details) we obtain the following combine operation:

$$
\begin{aligned}
&(mss\,x, mis\,x, mcs\,x, ts\,x) \;\circledast\; (mss\,y, mis\,y, mcs\,y, ts\,y) \;= \\
&\quad (\; mss\,x \uparrow (mcs\,x + mis\,y) \uparrow mss\,y \;,\; mis\,x \uparrow (ts\,x + mis\,y)\;, \\
&\quad\;\; mcs\,y \uparrow (mcs\,x + ts\,y)\;,\; (ts\,x + ts\,y)\;)
\end{aligned}
$$

Since $\circledast$ is associative, our tuple is the homomorphism: $<mss, mis, mcs, ts>= hom\,(f, \circledast)$, where f determines the result of the tuple on singleton list:

$$f\,(a) \;= <mss, mis, mcs, ts>\; [a] \;=\; (a \uparrow 0\,,\; a \uparrow 0\,,\; a \uparrow 0\,,\; a)$$

The target function *mss* is therefore computable as follows:

$$mss = fst \circ red\,(\circledast) \circ (map\,f)$$

If both function f and operation $\circledast$ require constant time, the total time complexity of this homomorphic algorithm is $O(\log n)$. The processor number can be reduced to $O(n/\log n)$ by simulating lower levels of the tree sequentially, based on Brent's theorem [25]. Therefore, the algorithm is both time and cost optimal.

In [12], we apply our "*cons* + *snoc* with generalization" method also to the parsing problem for so-called input-driven languages [8].

4 Concatenating and Distributable Homomorphisms

In this section, we address the second problem of the paper, finding an efficient parallel implementation for a given homomorphism.

The well-known difficulty arises when the output of a list homomorphism is a list again. In this case, the combine term has $+\!\!+$ as its top function: $com\text{-}op\,(u, v) = f\,(u, v) +\!\!+ g\,(u, v)$; we call such homomorphisms *concatenating*. The reduce stage starts from the singleton lists after the map stage and arrives at a "long" result list at the root of the tree. The communication of lists of growing length induces linear execution time, independently of the number of processors [27].

From (5) follows that *scan* is a concatenating homomorphism. However, there exist parallel logarithmic algorithms for *scan* with good performance on parallel machines [25]: rather than producing a monolithic output list, they distribute it between processors. Our goal is to derive such algorithms systematically.

From now on, we restrict ourselves to *powerlists* [21] of length 2^k, $k = 0, 1, \ldots$ with *balanced* concatenation and reduction: $x +\!\!+ y$ and $red\,(\odot)\,(x +\!\!+ y)$ are defined iff $length\,x = length\,y = 2^k$.

Definition 9. Distributable combine operation, $\oplus\!\bowtie\!\otimes$, on lists x and y of equal length:

$$x \oplus\!\bowtie\!\otimes\ y \;=\; zip\,(\oplus)\,(x,y) \;+\!\!+\; zip\,(\otimes)\,(x,y) \qquad (6)$$

where $\oplus$ and $\otimes$ are arbitrary binary associative operations on elements.

Definition 10. *Distributable homomorphism* (DH), denoted $(\oplus\updownarrow\otimes)$ for associative operations $\oplus$ and $\otimes$, is the unique homomorphism: $\oplus\updownarrow\otimes \;=\; \mathrm{hom}([.], \oplus\!\bowtie\!\otimes)$ with $\oplus\!\bowtie\!\otimes$ defined by (6).

Figure 1 illustrates, how DH is computed on a concatenation of two lists; dashed arrows denote replication of the partial results.

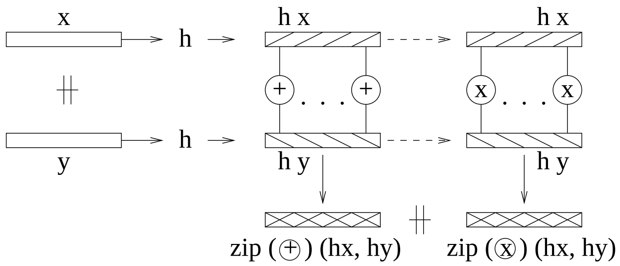

Fig. 1. Distributable homomorphism: an illustration

As a simple example, let us consider the function *sumd*: for a list of numbers $x = [x_1, x_2, \ldots, x_n]$, it yields: $sumd\ x = [sum\,x, sum\,x, \ldots, sum\,x]$, where $sum\,x = x_1 + x_2 + \cdots + x_n$ yields the sum of the list elements. It is easy to express *sumd* in the component-wise format (6):

$$sumd\,(x +\!\!+ y) = zip\,(+)\,(sumd\,x, sumd\,y) \;+\!\!+\; zip\,(+)\,(sumd\,x, sumd\,y)$$

Hence, $sumd \;=\; +\updownarrow+$. Generally, we will use the function called "distributed reduction", defined as $redd\,(\odot)\,x \;=\; [red\,(\odot)\,x, red\,(\odot)\,x, \ldots, red\,(\odot)\,x]$:

$$redd\,(\odot) \;=\; \odot\updownarrow\odot \qquad (7)$$

This function is implemented as `AllReduce` primitive in the MPI standard [30].

5 Towards an Efficient Parallel Implementation

Our ultimate goal is to find a provably correct and efficient parallel implementation for all DH functions. In this section, we first design an architecture-independent implementation schema and then map it onto hypercube topology. For that, we introduce some auxiliary functions on powerlists.

Our first two functions do simple rearrangements:

$$\begin{array}{ll} att\,:\, nat \to \alpha \to (nat, \alpha) & glue\,:\, \alpha \to (nat,\alpha) \to (nat,\alpha,\alpha) \\ att\ i\ x \;=\; (i, x) & glue\ a\,(i,b) \;=\; (i,\ a,\ b) \end{array}$$

Function *permute* interchanges pair-wise elements which have a given distance between their positions in the list, (the distance is the first argument of *permute*). The function attaches to each element a flag which is equal to 0, if the element has changed its position to the left and 1, otherwise:

$$
\begin{array}{lll}
\mathit{permute} : \mathit{nat} \rightarrow [\alpha] \rightarrow [(\mathit{nat}, \alpha)] & & \\
\mathit{permute}\, k\, (x +\!\!+\, y) \;=\; \mathit{permute}\, k\, x \;+\!\!+\; \mathit{permute}\, k\, y\,, & & \text{if } (k < \mathit{length}\,(x)) \\
\mathit{permute}\, k\, (x +\!\!+\, y) \;=\; \mathit{map}\,(\mathit{att}\, 0)\; y \;+\!\!+\; \mathit{map}\,(\mathit{att}\, 1)\; x\,, & & \text{if } (k = \mathit{length}\,(x))
\end{array}
$$

Function *triples* composes a list and the result of its permutation together:

$$
\begin{array}{l}
\mathit{triples} : \mathit{nat} \rightarrow [\alpha] \rightarrow [(\mathit{nat}, \alpha, \alpha)] \\
\mathit{triples}\, k\, x \;=\; \mathit{zip}\,(\mathit{glue})\,(x,\; \mathit{permute}\, k\, x)
\end{array}
$$

Function *apply* performs one of two binary operations ($\oplus$ or $\otimes$) on the elements of a list of triples, depending on the value of the flag:

$$
\begin{array}{l}
\mathit{apply} : ((\alpha \rightarrow \alpha \rightarrow \alpha), (\alpha \rightarrow \alpha \rightarrow \alpha)) \rightarrow (\mathit{nat},\, \alpha,\, \alpha) \rightarrow \alpha \\
\mathit{apply}\,(\oplus, \otimes)\,(i,\; a,\; b) \;=\; \text{if } \;(i = 0)\; \text{ then } \;(a \oplus b)\; \text{ else } \;(b \otimes a)
\end{array}
$$

Figure 2 illustrates how function *permute* and the next introduced function, *step*, work on a 4-element list.

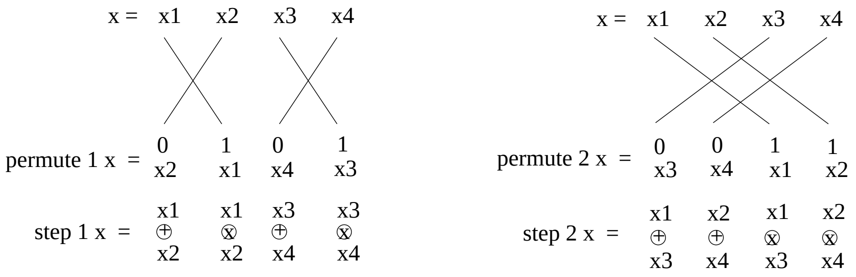

Fig. 2. Functions *permute* and *step*: an illustration

$$
\begin{array}{l}
\mathit{step} : \mathit{nat} \rightarrow ((\alpha \rightarrow \alpha \rightarrow \alpha), (\alpha \rightarrow \alpha \rightarrow \alpha)) \rightarrow [\alpha] \rightarrow [\alpha] \\
\mathit{step}\; k\; (\oplus, \otimes) \;=\; \mathit{map}\,(\mathit{apply}\,(\oplus, \otimes)) \circ (\mathit{triples}\; k)
\end{array}
$$

The following function, *iter*, does a sequence of *step* applications:

$$
\begin{array}{ll}
\mathit{iter} : \mathit{nat} \rightarrow \mathit{nat} \rightarrow ((\alpha \rightarrow \alpha \rightarrow \alpha), (\alpha \rightarrow \alpha \rightarrow \alpha)) \rightarrow [\alpha] \rightarrow [\alpha] & \\
\mathit{iter}\, k\, j\; (\oplus, \otimes) \;=\; \mathit{id}\,, & \text{if } (k = j) \\
\mathit{iter}\, k\, j\; (\oplus, \otimes) \;=\; (\, \mathit{iter}\,(2 * k)\; j\; (\oplus, \otimes)\,) \circ (\, \mathit{step}\; k\; (\oplus, \otimes)\,)\,, & \text{if } (k < j)
\end{array}
$$

The definition of *iter* is *tail-recursive*, with an obvious iterative implementation, rather than the usual *cascading recursion* in a homomorphism. The following theorem establishes the equivalence of these two forms.

Theorem 11. *For arbitrary associative $\oplus$ and $\otimes$ and lists of length 2^k:*

$$\oplus \updownarrow \otimes \;=\; \mathit{iter}\; 1\; k\; (\oplus, \otimes) \tag{8}$$

Proof: by induction on k.

The theorem provides a common iterative computation schema for all DH functions. The next step is to map this architecture-independent solution onto a particular processor topology. As an example, let us consider the hypercube.

Our lists of length $n = 2^k$ are stored in a k-dimensional hypercube with n nodes. The standard encoding is used: the position i, $0 \le i < n$, of a list is stored in the node i, whose index is the k-bit binary representation of i.

The access function on hypercube:

$$\mathit{hyp} : [\alpha] \to \mathit{nat} \to \alpha$$

yields, for list x and index i, the ith element of x.

Each processor of the hypercube can communicate directly with its k neighbours, whose indices differ in one bit position; this position determines the dimension, in which the communication takes place. In each dimension, $n/2$ pairs of processors can communicate simultaneously, without dilation or congestion. For processor i, its partner in the dimension $d = 1, 2, \ldots, k$ is computed as $p\,(i, d) = \mathit{xor}(i, 2^{d-1})$, where xor is "bit-wise exclusive OR".

Definition 12. Function *swap* expresses a pattern of the hypercube behaviour:

$$\mathit{swap} : \mathit{nat} \to ((\alpha \to \alpha \to \alpha), (\alpha \to \alpha \to \alpha)) \to [\alpha] \to [\alpha]$$

$$\mathit{hyp}\ (\mathit{swap}\ d\ (\oplus, \otimes)\ x)\ \ i \;=\; \begin{cases} (\mathit{hyp}\ x\ i) \oplus (\mathit{hyp}\ x\ (p(i,d)))\,, & \text{if } i < p(i,d) \\ (\mathit{hyp}\ x\ (p(i,d))) \otimes (\mathit{hyp}\ x\ i)\,, & \text{otherwise} \end{cases}$$

$$\text{where } \mathit{length}\,(x) = 2^k\,,\ p(i,d) = \mathit{xor}(i, 2^{d-1})\,,\ 1 \le d \le k\,,\ 0 \le i < 2^k.$$

From the definition follows that to compute the result of *swap* in processor i, this processor must access the element in position $p(i, d)$, i.e., communicate with its neighbour in dimension d. Thus, *swap* consists of pair-wise, two-directional communication in one dimension, followed by computation.

The following proposition establishes the correspondence between one step of the iterative solution (8) and one application of *swap*.

Theorem 13. *For lists of length* 2^k *and* $1 \le d \le k$, *holds:*

$$\mathit{step}\ (2^{d-1})\ (\oplus, \otimes) \;=\; \mathit{swap}\ d\ (\oplus, \otimes) \tag{9}$$

Introducing the notation:

$$\mathit{swap}^k\,(\oplus, \otimes) \;=\; (\mathit{swap}\,k\,(\oplus, \otimes)) \circ \cdots \circ (\mathit{swap}\,2\,(\oplus, \otimes)) \circ (\mathit{swap}\,1\,(\oplus, \otimes))$$

we obtain from Theorem 11 and Theorem 13 the following:

Corollary 14 (Common Hypercube Implementation). *Every DH can be computed on the* 2^k*-node hypercube by a sequence of swaps, with the dimensions counting from 1 to k:*

$$\oplus \updownarrow \otimes \;=\; \mathit{swap}^k\,(\oplus, \otimes) \tag{10}$$

Schema (10) expresses a standard way of programming hypercubes; its implementation as the target SPMD program with explicit message passing can be generated easily.

6 Implementation of Scan

In this section, we derive a parallel program that computes the *scan*-function.

Let us first check whether the *scan* function is DH. Its combine operator:

$$scan\,(\odot)\,(x \mathbin{+\!\!+} y) \;=\; S_1 \circledast S_2 \;=\; S_1 \mathbin{+\!\!+} map\;((last\,S_1)\odot)\;S_2, \qquad (11)$$
$$\text{where } S_1 = scan\,(\odot)\,x\,,\; S_2 = scan\,(\odot)\,y.$$

Our task is to express the right-hand side of (11) in the component-wise format (6), with both sides of $\mathbin{+\!\!+}$ in the *zip*-form. The part on the left of $\mathbin{+\!\!+}$: $S_1 = zip\,(\pi_1)\,(S_1, S_2)$, where π_1 yields the first element of a pair.

The obvious way to express the part on the right of $\mathbin{+\!\!+}$ component-wise is to "replicate" the element $last\,(S_1)$. Since $last\,(scan\,(\odot)\,x) = red\,(\odot)\,x$, the replication yields $(redd\,(\odot)\,x)$. This allows us to reformulate $\circledast$ component-wise:

$$S_1 \circledast S_2 \;=\; zip\,(\pi_1)\,(S_1, S_2) \mathbin{+\!\!+} zip\,(\odot)\,(R_1, S_2) \qquad (12)$$

where the introduced function, *redd*, is used: $R_1 = redd\,(\odot)\,x$.

Like for almost-homomorphisms above, we "tuple" both functions together: $< scan\,(\odot),\; redd\,(\odot) >$. To fit the DH format, we massage this tuple into a new function, *scred*, which yields a list of pairs instead of pair of lists:

$$scred\,(\odot) \;=\; zip\,(\diamond) \circ < scan\,(\odot),\; redd\,(\odot) > \qquad (13)$$

where $a \diamond b = (a, b)$, for elements a and b.

Since *redd* itself matches the DH format (7) with the combine operator:

$$R_1 \ominus R_2 \;=\; zip\,(\odot)\,(R_1, R_2) \mathbin{+\!\!+} zip\,(\odot)\,(R_1, R_2) \qquad (14)$$

no additional auxiliary functions are necessary.

Function *scred* can be expressed as follows:

$$scred\,(\odot) \;=\; (\oplus \updownarrow \otimes) \circ map\,(pair)\,, \qquad (15)$$

where function *pair* transforms an element into a pair; operations $\oplus$ and $\otimes$ work on pairs of elements $s_i \in S_i,\; r_i \in R_i$ and are directly read off from (12), (14):

$$\begin{aligned} pair\,a &= (a, a) \\ (s_1, r_1) \oplus (s_2, r_2) &= (s_1\,,\; r_1 \odot r_2) \\ (s_1, r_1) \otimes (s_2, r_2) &= (r_1 \odot s_2\,,\; r_1 \odot r_2) \end{aligned} \qquad (16)$$

From (13)-(15) follows the expression of *scan*, adjusted to the DH format:

$$scan\,(\odot) \;=\; (map\,\pi_1) \circ (\oplus \updownarrow \otimes) \circ (map\;pair) \qquad (17)$$

where *pair*, $\oplus$, $\otimes$ are defined by (16).

Thus, we have adjusted the *scan* function to the DH format. The tuple structure, its initialization by function *pair*, and the computations expressed by $\oplus$ and $\otimes$ in (16), are all the results of the systematic adjustment process.

We can now directly rewrite (17) by using the implementation schema (10) and thus obtain the hypercube program for *scan*:

$$scan\,(\odot) \;=\; (map\ \pi_1)\ \circ\ swap^k\,(\oplus,\otimes)\ \circ\ (map\ pair\,) \tag{18}$$

with *pair*, $\oplus$ and $\otimes$ from (16).

This is the well-known "folklore" implementation [25]. In Figure 3, it is illustrated for the 2-dimensional hypercube which is computing $scan\,(+)\,[1,2,3,4]$.

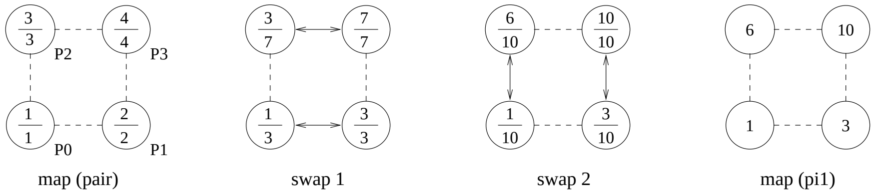

Fig. 3. Computing *scan* on a hypercube

Implementation (18) consists of three stages: pairing, repeating swaps and projecting. For a list of length $n = 2^k$, we need n processors. Both the pairing and the projection stage require constant time. The central stage is the sequential loop with k swaps; in each swap, pairs of elements are communicated and computations are performed on pairs as well, so every swap requires constant time. Hence, the time is $O\,(k) \;=\; O\,(\log n)$, our implementation is *time optimal*.

The cost (time-processor product [25]) is $O\,(n \log n)$, whereas the cost of the sequential computation is $O\,(n)$, so the implementation is *not cost optimal*. To improve it, we must use fewer processors, with each processor working with a segment of the input list. Formal derivation of a time and cost optimal algorithm for this practical situation exploits BMF-transformations which involve data distributions; this is the subject of another paper [14].

7 Related Work

Our approach to parallelism extraction can be compared with [15] since we consider the same examples. On simple examples like *length*, we actually do the same. For *scan*, rather involved calculations and intuition are required in [15] to obtain $\circledast$; here the advantages of the systematic CS-approach are evident. For almost-homomorphisms, the method of [15] is not suitable at all. In [2], the existence of leftwards and rightwards algorithms is used as an evidence that a homomorphic algorithm exists; unlike our approach, the authors do not provide a method to derive it.

Our solution for the maximum segment sum problem is similar to those provided earlier by Smith [28] and Cole [7]. Our contribution is the systematic CS-method which, first, provides a uniform way of introducing the necessary auxiliary functions and, second, exploits a rigorous generalization procedure for deriving the resultant combine operation on tuples.

Parallelization of the *scan* function has a rich history, starting from the seminal work by Ladner and Fisher [20]. Meanwhile, parallel algorithms for *scan* are a part of folklore [25], and are usually presented in an *ad hoc* manner.

There are few exceptions, to which we compare our approach. Mou [6, 22] specifies the *scan* algorithm within an algebraic model of divide-and-conquer and suggests an optimization, which is similar to ours; the tuple structure arises by a non-formal argument and the result is not proved formally. A tree algorithm has been verified formally by O'Donnell [23], and later derived formally by J. Gibbons [10]. Kornerup [17] arrives formally at the algorithm by Ladner and Fisher in the recursive powerlist notation. Our approach differs in that our target implementation (1) is a result of the systematic, provably correct adjustment and specialization process and (2) is obtained in an iterative form, where all stages of computations and communication can be seen explicitly.

The construction of function *iter* is a special case of the compound list operations by Kumar and Skillicorn [19], which we use here for a different purpose. Our restriction to lists of length 2^k has also much in common with the powerlists by Misra [21]. Unlike him, we get rid of the explicit recursion in the target program by introducing iterative constructs. However, our approach is more restrictive, since we do not consider the list interleaving constructor $\bowtie$ used by Misra.

An approach similar to ours in deriving an architecture-independent solution and mapping it onto particular topologies has been taken by Achatz and Schulte [1]. We consider a more special class DH, which allows us to exploit additional transformations. Our approach can be extended to an arbitrary number of processors [14], unlike the SIMD model used in [1].

The general implementation schema for DH functions on the hypercube resembles the common structure of *ascending* algorithms studied in the seminal paper by Preparata and Vuillemin [24]. We view this analogy is a promising sign for research towards building a taxonomy of functions with respect to their efficient parallel implementations.

8 Conclusion and Future Work

We propose an approach to exploiting divide-and-conquer parallelism in functions on lists, which consists of two steps: first, parallelism extraction by finding a homomorphic representation of the given function, second, parallelism implementation by adjusting the function to the DH format and using the common parallel implementation schema.

We claim that our approach is more systematic than the methods presented previously, for the following reasons:

- At the *parallelism extraction* step, the user provides two sequential definitions of a given function in a closed leftwards and rightward form. According to Theorem 4, a difficulty in finding an *lw* or *rw* form indicates that the function might be non-homomorphic. The requirement of closeness, as we see in the *mss* example, "guides" the introduction of the necessary auxiliary functions. The rest of the job is done by the generalization procedure.

– At the *parallelism implementation* step, the function must be cast in the DH format, which again serves as a guide for the user. After that, the implementation schema is customized correspondingly.

Methodologically, an important feature of the parallelism extraction step is that it is based on sequential thinking: the developer is required to provide two sequential functional programs, which are then transformed by the generalization procedure. Considerations involving data and control dependences, which are usual in parallelization techniques, are completely avoided.

The contribution to the implementation methodology is in the definition of the DH class of functions on lists and the formal derivation of a common efficient parallel implementation schema for all functions of the class. The derivation is based on the semantically sound transformation rules of the BMF, which guarantees its correctness. The performance of the common target implementations is easily predictable and conforms with the known estimates.

Our future work includes designing a standard generalization procedure for the CS method and also extending a class of functions for which efficient parallel implementation schemata for various architectures can be built systematically.

9 Acknowledgments

I am grateful to Murray Cole, Alfons Geser, Jeremy Gibbons, Christian Lengauer, Lambert Meertens and Christoph Wedler for discussing different parts of the manuscript. The anonymous referees helped a lot to make it a better paper.

The author was partially supported by the DAAD cooperation programs ARC and PROCOPE, and by the Project INTAS-93-1702.

References

1. K. Achatz and W. Schulte. Architecture independent massive parallelization of divide-and-conquer algorithms. In B. Moeller, editor, *Mathematics of Program Construction*, Lecture Notes in Computer Science 947, pages 97–127, 1995.
2. D. Barnard, J. Schmeiser, and D. Skillicorn. Deriving associative operators for language recognition. *Bulletin of EATCS*, 43:131–139, 1991.
3. J. Bentley. Programming pearls. *Communications of the ACM*, 27:865–871, 1984.
4. R. S. Bird. Lectures on constructive functional programming. In M. Broy, editor, *Constructive Methods in Computing Science*, NATO ASO Series F: Computer and Systems Sciences. Vol. 55, pages 151–216. Springer Verlag, 1988.
5. G. Blelloch. Scans as primitive parallel operations. *IEEE Trans. on Computers*, 38(11):1526–1538, November 1989.
6. B. Carpentieri and G. Mou. Compile-time transformations and optimizations of parallel divide-and-conquer algorithms. *ACM SIGPLAN Notices*, 20(10):19–28, 1991.
7. M. Cole. Parallel programming with list homomorphisms. *Parallel Processing Letters*, 5(2):191–204, 1994.
8. A. Gibbons and W. Rytter. *Efficient Parallel Algorithms.* Cambridge Univ. Press, 1988.

9. J. Gibbons. The third homomorphism theorem. *J. Fun. Programming*. To appear.
10. J. Gibbons. Upwards and downwards accumulations on trees. In R. Bird, C. Morgan, and J. Woodcock, editors, *Mathematics of Program Construction*, Lecture Notes in Computer Science 669, pages 122–138, 1992.
11. J. Gibbons. The third homomorphism theorem. Technical report, Univ. of Auckland, 1994.
12. S. Gorlatch. Constructing list homomorphisms. Technical Report MIP-9512, Universität Passau, 1995.
13. S. Gorlatch. Stages and transformations in parallel programming. In M. Kara et al., editors, *Abstract Machine Models for Parallel and Distributed Computing*, pages 147–162. IOS Press, 1996.
14. S. Gorlatch. Systematic optimal parallelization of scan and other list homomorphisms. In *Proceedings of the Euro-Par'96*. LNCS, to appear, 1996.
15. Z. Grant-Duff and P. Harrison. Parallelism via homomorphisms. *Parallel Processing Letters*. To appear.
16. B. Heinz. Lemma discovery by anti-unification of regular sorts. Technical Report 94-21, TU Berlin, May 1994.
17. J. Kornerup. Mapping a functional notation for parallel programs onto hypercubes. *Information Processing Letters*, 53:153–158, 1995.
18. H. Kuchen, R. Plasmeijer, and H. Stolze. Distributed implementation of a data-parallel functional language. In *PARLE'94, LNCS 817*, pages 464–477, 1994.
19. K. Kumar and D. Skillicorn. Data parallel geometric operations on lists. *Parallel Computing*, 21(3):447–459, 1995.
20. R. Ladner and M. Fischer. Parallel prefix computation. *J. ACM*, 27:831–838, 1980.
21. J. Misra. Powerlist: a structure for parallel recursion. *ACM TOPLAS*, 16(6):1737–1767, 1994.
22. Z. G. Mou. Divacon: A parallel language for scientific computing based on divide and conquer. In *Proc. 3rd Symposium on the Frontiers of Massively Parallel Computation*, pages 451–461, October 1990.
23. J. O'Donnell. A correctness proof of parallel scan. *Parallel Processing Letters*, 4(3):329–338, 1994.
24. F. Preparata and J. Vuillemin. The cube-connected cycles: A versatile network for parallel computation. *Communications of the ACM*, 24(5):300–309, 1981.
25. M. J. Quinn. *Parallel Computing*. McGraw-Hill, Inc., 1994.
26. D. Skillicorn. *Foundations of Parallel Programming*. Cambridge Univ. Press, 1994.
27. D. Skillicorn and W. Cai. A cost calculus for parallel functional programming. *Journal of Parallel and Distributed Computing*, 28:65–83, 1995.
28. D. Smith. Applications of a strategy for designing divide-and-conquer algorithms. *Science of Computer Programming*, (8):213–229, 1987.
29. D. Swierstra and O. de Moor. Virtual data structures. In B. Moeller, H. Partsch, and S. Schuman, editors, *Formal Program Development*, Lecture Notes in Computer Science 755, pages 355–371.
30. D. Walker. The design of a standard message passing interface for distributed memory concurrent computers. *Parallel Computing*, 20:657–673, 1994.